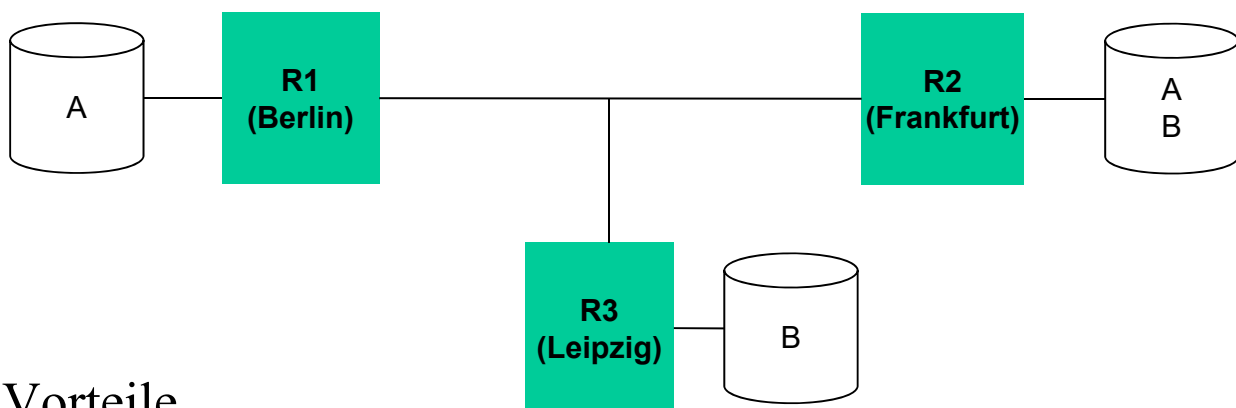


# 7. Replizierte Datenbanken

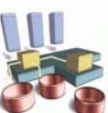
- Einführung (Replikationsstrategien)
- Update-Strategien bei strenger Konsistenz
  - ROWA-Verfahren / 2PC
  - Voting-Verfahren
- schwächere Formen der Datenreplikation
  - Refresh-Alternativen
  - Primary-Copy-Verfahren
  - Schnappschuß-Replikation
  - Merge-Replikation
  - Replikationsansätze in kommerziellen DBS (MS SQL-Server)
- Katastrophen-Recovery
- Geo-Replikation für Cloud-Daten
- Datenreplikation in Parallelen DBS
  - verstreute Replikation
  - verkettete Replikation



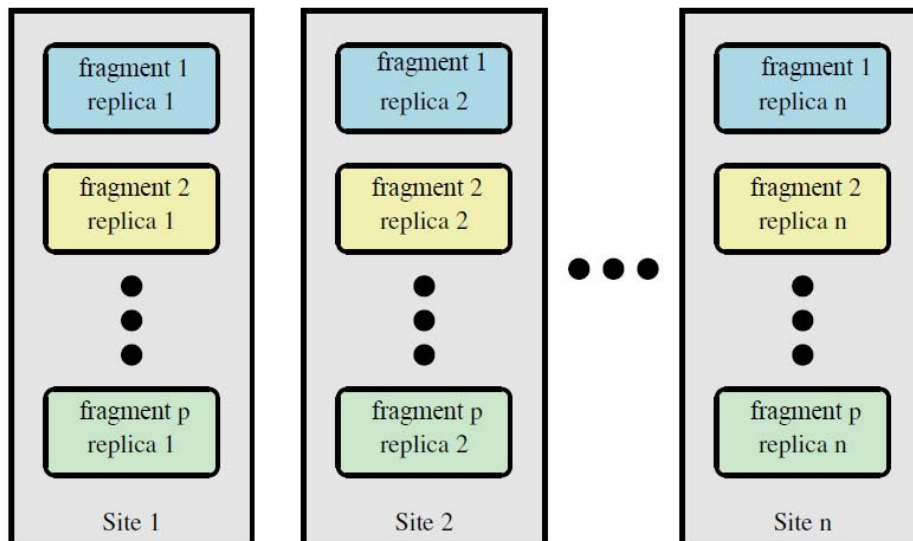
## Replikate in Verteilten DBS



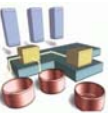
- Vorteile
  - erhöhte Verfügbarkeit der Daten
  - Beschleunigung von Lesezugriffen (bessere Antwortzeiten, Kommunikationseinsparungen)
  - erhöhte Möglichkeiten zur Lastbalancierung / Query-Optimierung
- Nachteile
  - hoher Update-Aufwand
  - erhöhter Speicherplatzbedarf
  - erhöhte Systemkomplexität



# Partitionierte und replizierte DBS

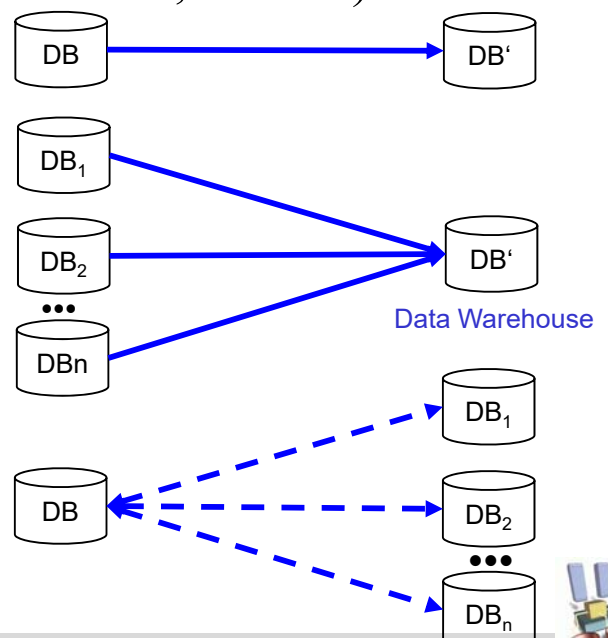


- n-fache Replikation jeder DB-Partition (Shared Nothing)
- häufige Verwendung auch in NoSQL-DBS
- ggf. Geo-Replikation (ortsverteilte Server)



## Replikation: Anwendungsmöglichkeiten

- Verteilte und Parallele DBS
  - replizierte DB-Tabellen / -Fragmente
  - replizierte Katalogdaten (Metadaten)
- Web: replizierte Daten und Metadaten für schnelles Lesen und Verfügbarkeit (Mirror Sites, Suchmaschinen, Portale)
- Katastrophen-Recovery

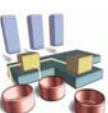


- Data Warehousing

- Übernahme transformierter Daten in eigene Datenbank für Entscheidungsunterstützung

- Mobile Computing

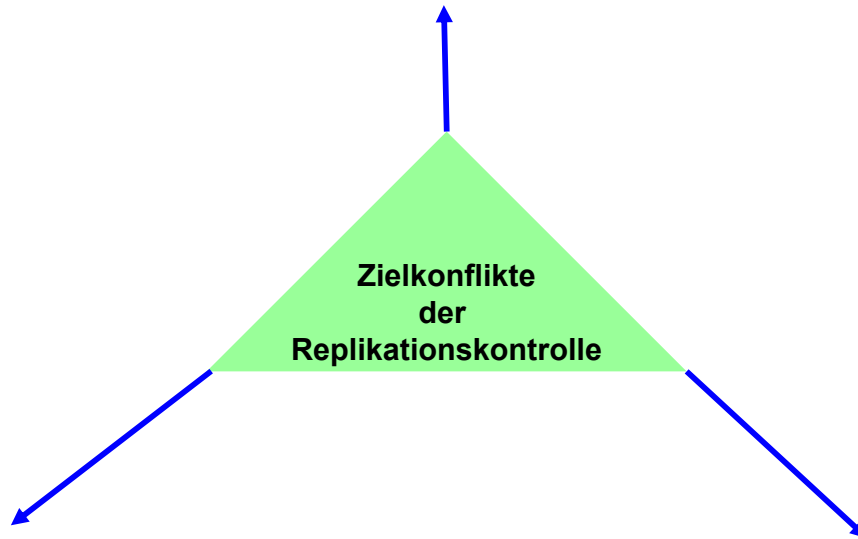
- Datenbank-Ausschnitte, Dateien ... auf Notebook, PDA etc.



# Zielkonflikte der Replikationskontrolle

## Erhaltung der Datenkonsistenz

- Kopien wechselseitig konsistent zu halten: *1-Kopien-Äquivalenz*
- kleine Kopienzahl



## Erhöhung der Verfügbarkeit, effizienter Lesezugriff

- große Kopienzahl
- Zugriff auf beliebige und möglichst wenige Kopien

## Minimierung des Änderungsaufwands

- kleine Kopienzahl
- möglichst wenige Kopien synchron aktualisieren



## Replikationsstrategien (1)

### ■ Korrektheitskriterium / Synchronisation

### ■ 1-Kopien-Äquivalenz:

- vollständige Replikationstransparenz: jeder Zugriff liefert jüngsten transaktionskonsistenten Objektzustand
- alle Replikate sind wechselseitig konsistent
- Serialisierung von Änderungen

### ■ geringere Konsistenzanforderungen

- Zugriff auf ältere Objektversionen
- evtl. keine strikte Serialisierung von Änderungen, sondern parallele Änderungen mit nachträglicher Konflikt-Behebung



# Replikationsstrategien (2)

- symmetrische vs. asymmetrische Konfigurationen
  - **symmetrisch**: jedes Replikat gleichartig bezüglich Lese- und Schreibzugriffen; n Knoten können Objekt ändern
  - **asymmetrisch** (Master/Slave, **Publish/Subscribe**-Ansätze)
    - Änderungen eines Objekts an 1 Knoten (Master, Publisher, Primärkopie)
    - Lesen an n Knoten (Slave, Subscriber, Sekundärkopien)
  - **Kombinationen** (z.B. Multi-Master-Ansätze)
  
- Zeitpunkt der Aktualisierung: synchron oder asynchron
  - **synchron** (eager): alle Kopien werden durch Änderungstransaktion aktualisiert
  - **asynchron** (lazy): verzögertes Aktualisieren (Refresh)



## Grobklassifikation

### Replikationsstrategien

#### Korrektheitsziel

strenge Konsistenz  
(1-Kopien-Serialisierbarkeit)

schwache Konsistenz  
(weak consistency)

#### Refresh-Zeitpunkt

synchron  
(eager)

asynchron  
(lazy)

Asynchron (lazy)

#### #Änderer (Masters) pro Objekt

n

1

n

1

#### Beispiel-Strategien

- ROWA  
(2PC)  
- Voting

- Primary Copy  
(mit Lesen  
aktueller Daten)

- Merge-Replikation  
(„Update Anywhere“)

- Primary Copy mit  
Lesen veralteter  
Daten  
- Schnappschuß-  
Replikation  
- Standby-Replikation



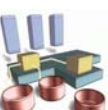
# Write-All/Read-Any-Strategie (Read-One/Write-All, ROWA)

- bevorzugte Behandlung von Lesezugriffen
  - Lesen eines beliebigen (lokalen) Replikats
  - hohe Verfügbarkeit für Leser
- sehr hohe Kosten für Änderungstransaktionen
  - Schreibsperrern bei allen Replikaten anfordern
  - Propagierung der Änderungen im Rahmen des Commit-Protokolls (2PC)
- **Verfügbarkeitsproblem:** Änderungen von Verfügbarkeit aller kopienhaltenden Knoten abhängig
  - sei  $p$  die mittlere Wahrscheinlichkeit, daß ein Knoten verfügbar ist
  - bei  $N$  Kopien beträgt die Wahrscheinlichkeit, daß *geschrieben* werden kann:  $p^N$
  - Wahrscheinlichkeit, daß *gelesen* werden kann:  $1 - (1 - p)^N$



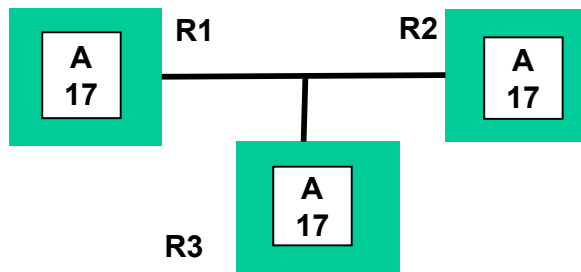
## Write All Available

- ROWA-Variante, bei der nur verfügbare Replikate geändert werden
  - für ausgefallene Rechner werden Änderungen bei Wiederanlauf nachgefahren
  - funktioniert nicht bei Netzwerk-Partitionierungen



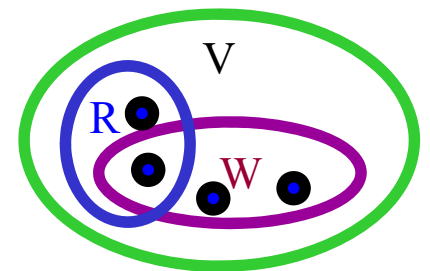
# Mehrheits-Votieren (Majority Voting)

- Lesen oder Schreiben eines Objektes verlangt Zugriff auf Mehrheit der Replikate
- jedes Replikat kann gleichzeitig von mehreren Transaktionen gelesen, jedoch nur von einer Transaktion geändert werden
- Bestimmung der aktuellsten Version z. B. über Änderungszähler
- Fehlerfall: Fortsetzung der Verarbeitung möglich, solange Mehrheit der Replikate noch erreichbar
- hohe Kommunikationskosten für Lese- und Schreibzugriffe
  - ungeeignet für  $N=2$  („Read All, Write All“)



# Gewichtetes Votieren (Quorum Consensus)

- jede Kopie erhält bestimmte Anzahl von Stimmen (votes)
- Protokoll:
  - Lesen erfordert  $R$  Stimmen (Read-Quorum)
  - Schreiben erfordert  $W$  Stimmen (Write-Quorum)
  - $R + W > V$  (= Summe aller Stimmen)
  - $W > V/2$



- Eigenschaften:
  - gleichzeitiges Lesen und Schreiben nicht möglich
  - jeder Zugriff auf  $R$  ( $W$ ) Kopien enthält wenigstens 1 aktuelle Kopie
  - Festlegung von  $V$ ,  $R$  und  $W$  erlaubt Trade-Off zwischen Lese- und Schreibkosten sowie zwischen Leistung und Verfügbarkeit

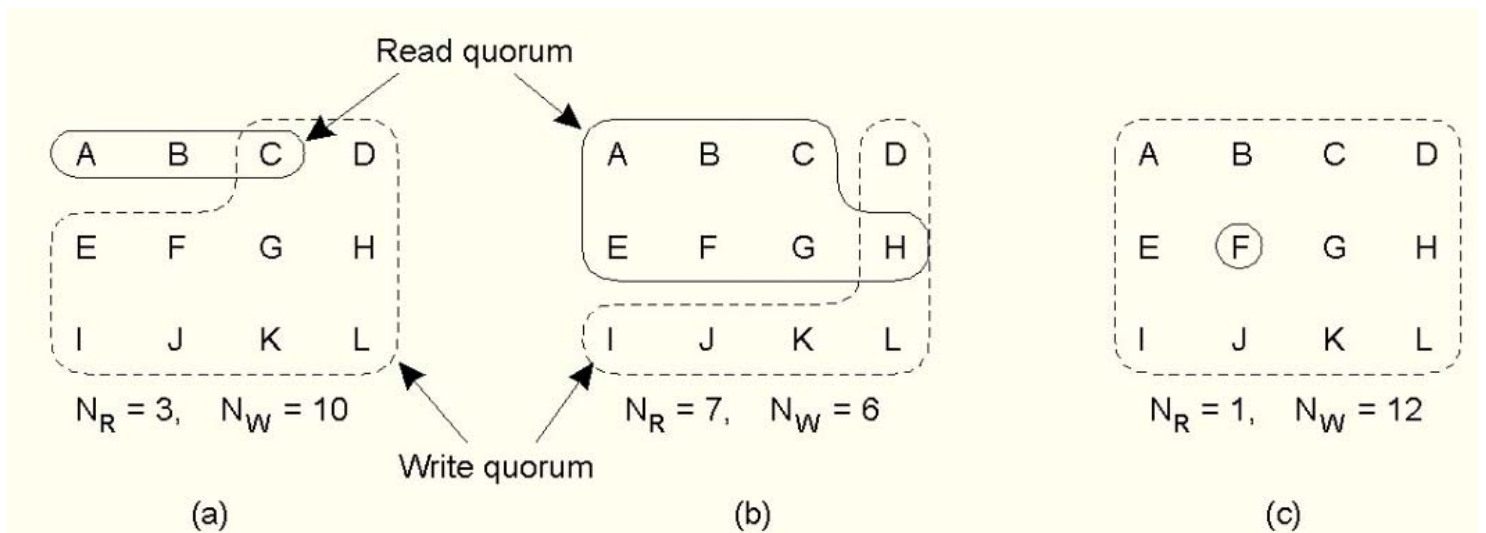


## Gewichtetes Votieren (2)

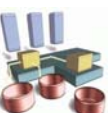
- andere Verfahren ergeben sich als Spezialfälle
  - Beispiel : 3 Kopien mit jeweils 1 Stimme ( $V=3$ )
  - ROWA:
  - Majority Voting (Consensus):



## Gewichtetes Votieren (3)



source: Tanenbaum and van Steen, Distributed Systems: Principles and Paradigms, Prentice-Hall, Inc. 2002



# schwächere Replikationsformen

- gravierende Probleme bei vollständiger Replikationstransparenz
  - hohe Änderungskosten bei synchroner Propagierung von Änderungen
  - Verfügbarkeitsprobleme (besonders bei geographischer Verteilung)
  - geringe Skalierbarkeit!
  - nicht anwendbar für mobile Replikate (meist offline)
- DBS-Hersteller (Oracle, IBM, MS) unterstützen schwächere Konsistenzformen mit asynchroner Aktualisierung der Kopien
  - Schnappschuß-Replikation
  - "fast" aktuelle Kopien, z.B. für Unterstützung einer schnellen Katastrophen-Recovery
- bei  $n$  Änderungsknoten pro Objekt („Update Anywhere“):
  - asynchrones Refresh führt zu Konsistenzproblemen
  - anwendungsspezifische Konflikt-Behebung (Merge-Replikation)



## Refresh-Alternativen

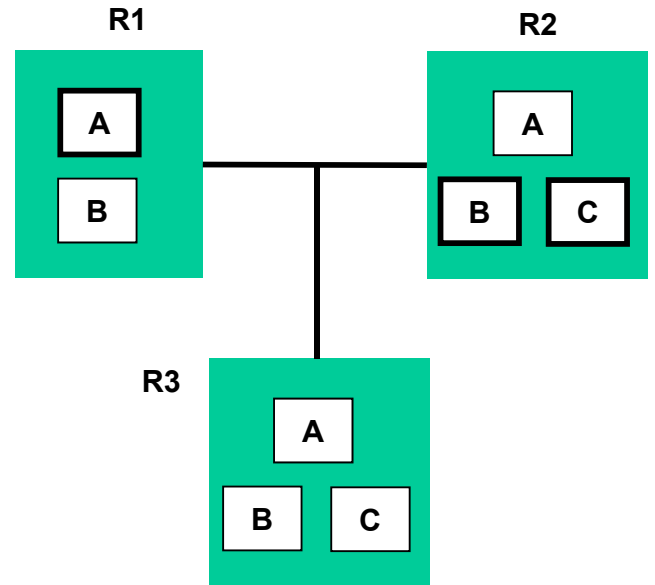
- baldmögliche Weiterleitung (ASAP) nach Commit der Änderungstransaktion oder verzögert (Timer, #Änderungen)
  - Verzögerungsdauer: Aktualität vs. Overhead
- **Push- vs. Pull-Replikation**
  - Push: Notifikation von Änderungen durch Master / Publisher
  - Pull: Probing durch Slave/Subsriber (z. B. mobiles Endgerät)
- **einfache Replikate (Kopien) vs. abgeleitete Replikation** (Ergebnisse einer SQL-Operation)
- **vollständige vs. inkrementelle Aktualisierung** von Replikaten
  - Übertragung der Objektkopien / Werte
  - Verwendung von Triggern
  - Übertragung der Log-Daten





# Primärkopien-Verfahren

- asymmetrisches Verfahren
  - 1 Primärkopie (primary copy):  
*publisher*
  - n Sekundärkopien pro Objekt:  
*subscriber*
- Bearbeitung von Schreib- und Lesesperren beim Primärkopien-Rechner
- synchrone Änderungen nur für Primärkopie
- verzögerte/asynchrone Aktualisierungen der Sekundärkopien
  - in der selben Commit-Reihenfolge wie bei Primärkopien-Rechner
  - Verwendung von Sequenz-Nummern



Die Primärkopien verschiedener Objekte können auf verschiedenen Rechnern liegen



## Primärkopien-Verfahren: Lesezugriffe

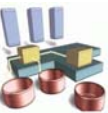
mehrere Alternativen

- Sperren und Lesen der Primärkopie
  - aktuelle Daten
  - Replikation wird nicht genutzt!
- lokale Kopie lesen ohne Sperre
  - effizient
  - lokale Kopie kann leicht veraltet sein (keine 1-Kopien-Serialisierbarkeit)
- Lesesperre beim Primärkopien-Rechner + Lesen einer aktuellen (lokalen) Kopie
  - z.B. Test über Versionsnummer bei Primärkopien-Rechner
  - strenge Konsistenz, jedoch geringe Kommunikationseinsparung



# Primärkopien-Verfahren: Fehlerbehandlung

- Netzwerk-Partitionierung: nur in Partition mit Primärkopie können weiterhin Änderungen erfolgen
- Ausfall des Primary-Copy-Rechners verhindert weitere Schreibzugriffe bis zu dessen Recovery
- Alternative: Bestimmung eines neuen Primary-Copy-Rechners, z.B. mit Paxos-Protokoll
  - jede Änderung erfordert, dass Mehrheit der Kopien nicht nur Primärkopie geändert wird
  - auch nach Ausfall der Primärkopie entsteht kein Datenverlust und Verarbeitung kann mit aktuellen Daten fortgesetzt werden



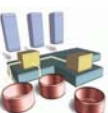
## Schnappschuß-Replikation

- Erzeugung einer materialisierten Sicht zur Nutzung an anderen Rechnern

### Beispiel:

```
CREATE SNAPSHOT CS-BOOKS AS
      SELECT *
      FROM BOOKS
      WHERE TOPIC=7
REFRESHED EVERY MONTH;
```

- Merkmale
  - Schnappschuß meist nicht auf dem neuesten Stand (-> reduzierter Änderungsaufwand)
  - i.a. nur lesender Zugriff (-> keine Synchronisationskonflikte auf Kopien)
  - Replikation für Benutzer i.a. nicht transparent



# Schnappschuß-Replikation (2)

- Aktualisierung der Kopie (Refresh)
  - periodisch (täglich, monatlich, etc.)
  - explizit auf Anforderung: **REFRESH SNAPSHOT CS\_BOOKS**
- Refresh-Optionen in Oracle (für Materialized Views)
  - Refresh Type: Complete, Fast (incremental), Force (system-determined)
  - Refresh Interval: on demand, on commit, automatically on, never
- zahlreiche Anwendungsmöglichkeiten
  - besonders geeignet für Datennutzung auf mobilen Geräten

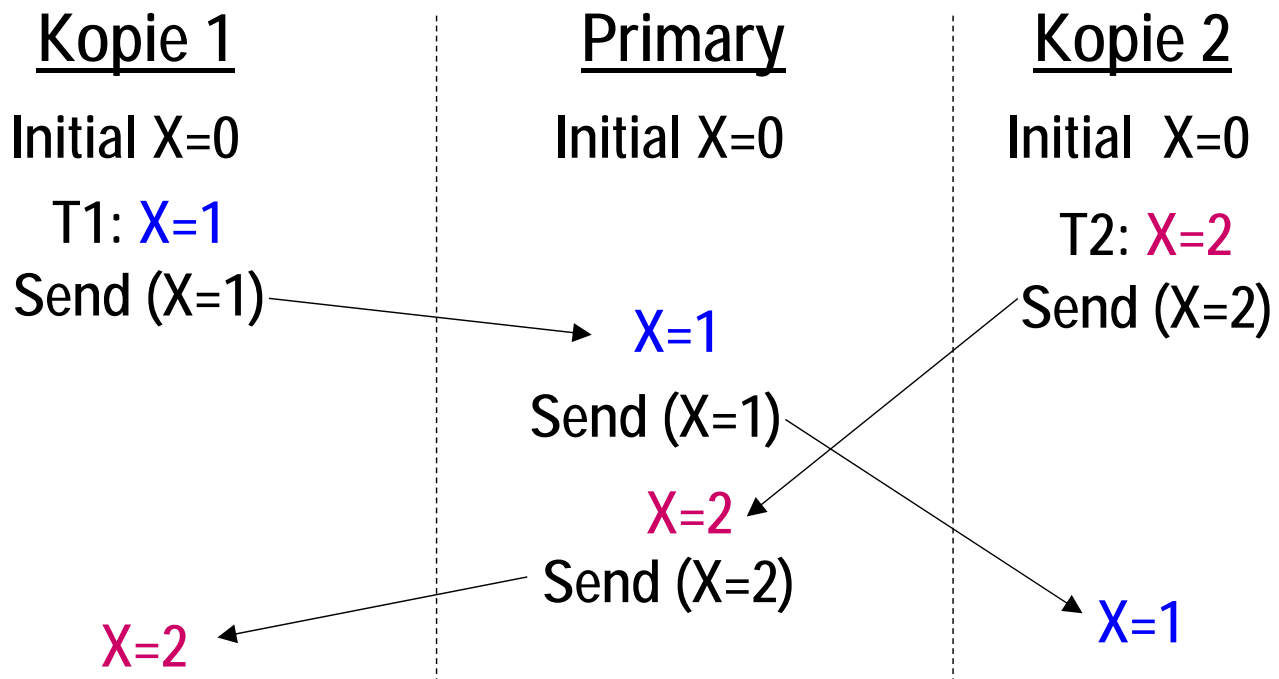


## Merge-Replikation

- unsynchronisierte Änderung eines replizierten Objekts an mehreren Knoten (Multi-Master) mit asynchroner Propagierung der Änderungen
- Performance- und Verfügbarkeitsvorteile gegenüber synchroner Aktualisierung
- unabdingbar für mobile DB-Nutzung mit Änderungsbedarf (z.B. Außendienst-Mitarbeiter)
  - Eintragung neuer Kunden / Aufträge (geringe Konfliktgefahr)
- Probleme:
  - nachträgliche Konflikt-Behandlung
  - Mischen paralleler Änderungen (Merge-Replikation)



# Beispiel für Änderungskonflikt bei unabhängiger Änderung



- Kopien enden in unterschiedlichen Zuständen



## Merge-Replikation (2)

### ■ Konfliktmöglichkeiten für

- Update
- Insert (z. B. Eindeutigkeit)
- Delete

### ■ Konflikterkennung

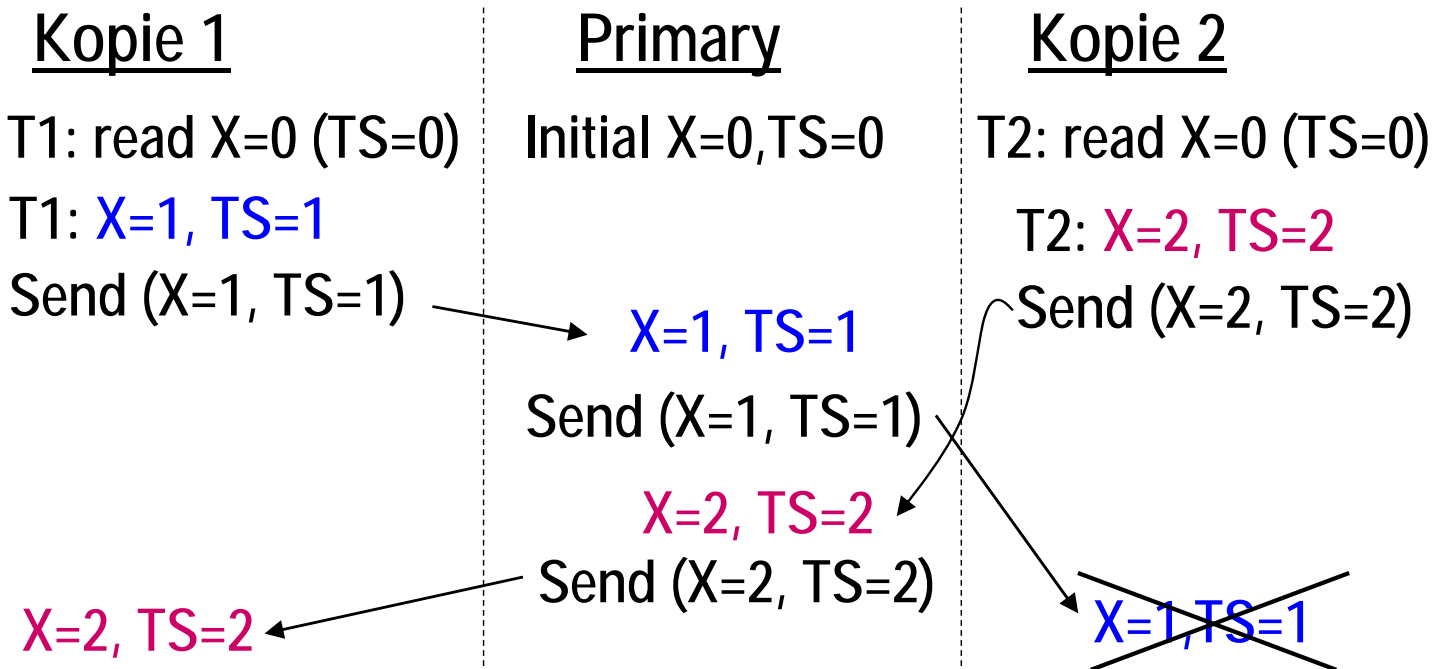
- Objektänderungen enthalten Zeitstempel  $v$  der Vorgängerversion und neuen Wert
- Konflikt: falls lokaler Zeitstempel einer zu aktualisierenden Objektversion abweicht von  $v$
- Konfliktwahrscheinlichkeit wächst mit Anzahl der änderbaren Replikate und Aktualisierungsverzögerung

### ■ anwendungsspezifische Konflikt-Behandlung (reconciliation)

- vordefinierte Strategien in ex. DBS:  
„last timestamp wins“, „site priority“, „average“ ...
- benutzerdefinierte Konfliktauflösungsroutinen oder manuelle Konfliktbehebung
- Gefahr von „lost updates“ und anderen Anomalien (keine Serialisierbarkeit)



# einfache Zeitstempelnutzung

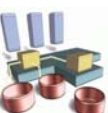


Kopien enden in gleichem Zustand, jedoch hat weder T1 noch T2 das Ergebnis der anderen Transaktion gesehen (nicht serialisierbar)



## Replikation in MS SQL-Server

- 3 Varianten mit asynchroner Aktualisierung: Snapshot, Transactional Replication, Merge Replication
- Snapshot Replication
  - Immediate vs. Queued Updating
- Transactional Replication
  - inkrementelle Aktualisierung durch transaktionsweise Übertragung und Anwendung von Log-Daten
  - asynchrone („bi-directional“) vs. Peer-to-Peer
  - Spezialfälle: 1 Publisher / n Subscriber vs. n Publisher / 1 Subscriber (Außendienst-Mitarbeiter - Zentrale)
- Merge Replication: mehrere Änderer pro Replikat
  - Einsatz von Triggern zur Propagierung von Änderungen
  - benutzerdefinierte Konfliktauflösungsstrategien (GUI-basiert)

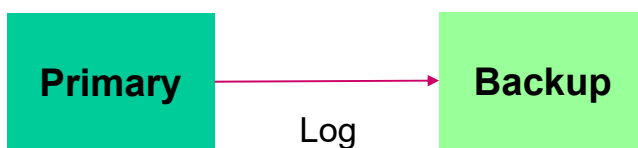


# Katastrophen-Recovery

- traditionell: Zurückgehen auf Archivkopie
  - Verlust vieler Transaktionen falls aktuelle Log-Daten seit Erstellung der Archivkopie verloren gehen
  - zeitaufwendiges Einspielen der Archivkopie
- Remote Logging: Übertragung der Log-Daten an entferntes Rechenzentrum
- Alternative für hohe Verfügbarkeit: räumlich entfernte Rechenzentren mit replizierter Datenbank
  - Replikationswartung gemäß Primary-Copy-Verfahren (Primär- und Sekundärkopien)
  - kommerzielle DBS: meist zwei Rechenzentren, davon nur in einem Änderungsbetrieb
  - Web Data (Google, Yahoo, Ebay, Amazon): mehrere (<10) aktive Data Center mit replizierten Daten



## Katastrophen-Recovery (2)



- passives Stand-By-System (Hot Stand-By)
  - Änderungen erfolgen nur im Primärsystem
  - Kopie dient als Stand-By bzw. wird nur für Queries genutzt (Entlastung des Primärsystems)
  - Redo-Log-Daten werden ständig zum Backup-System übertragen und DB-Kopie wird nachgefahren
  - bei asynchroner Übertragung gehen höchstens einige wenige Transaktionen im Katastrophenfall verloren
  - für "wichtige" Transaktionen: synchrone Übertragung der Log-Daten (Commit-Verzögerung bis entfernte DB aktualisiert ist)



# Katastrophen-Recovery: Sicherheitsstufen

## ■ 1-sicher (1-safe)

- lokales Commit am Primary
- asynchrone Übertragung der Änderungen an Backups

## ■ 2-sicher (2-safe)

- synchrone Aktualisierung der Backups
- Kommunikation in Commit-Phase 1

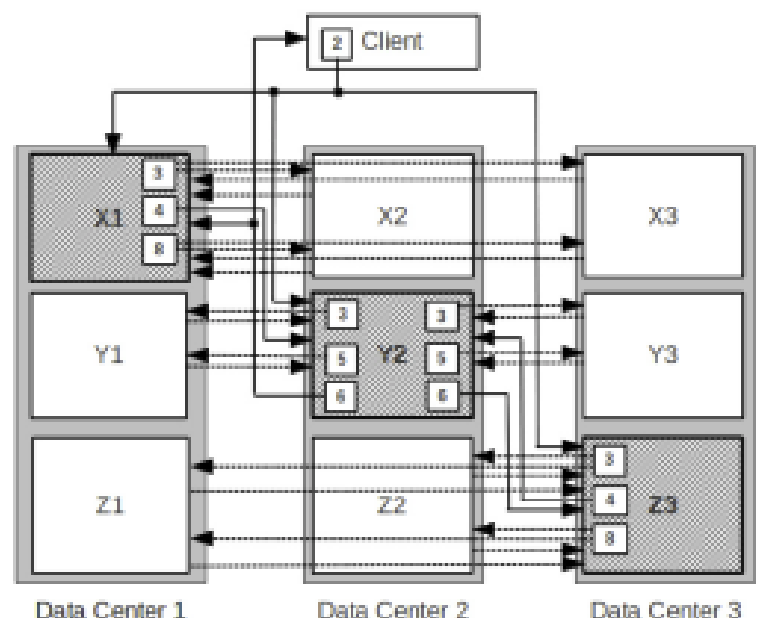
## ■ "sehr sicher"

- 2PC bzw. Paxos-Verfahren zwischen Primary und Backups
- alle Knoten (bzw. Mehrheit) müssen Commit zustimmen
- nur sinnvoll bei gleichberechtigten Data Centers

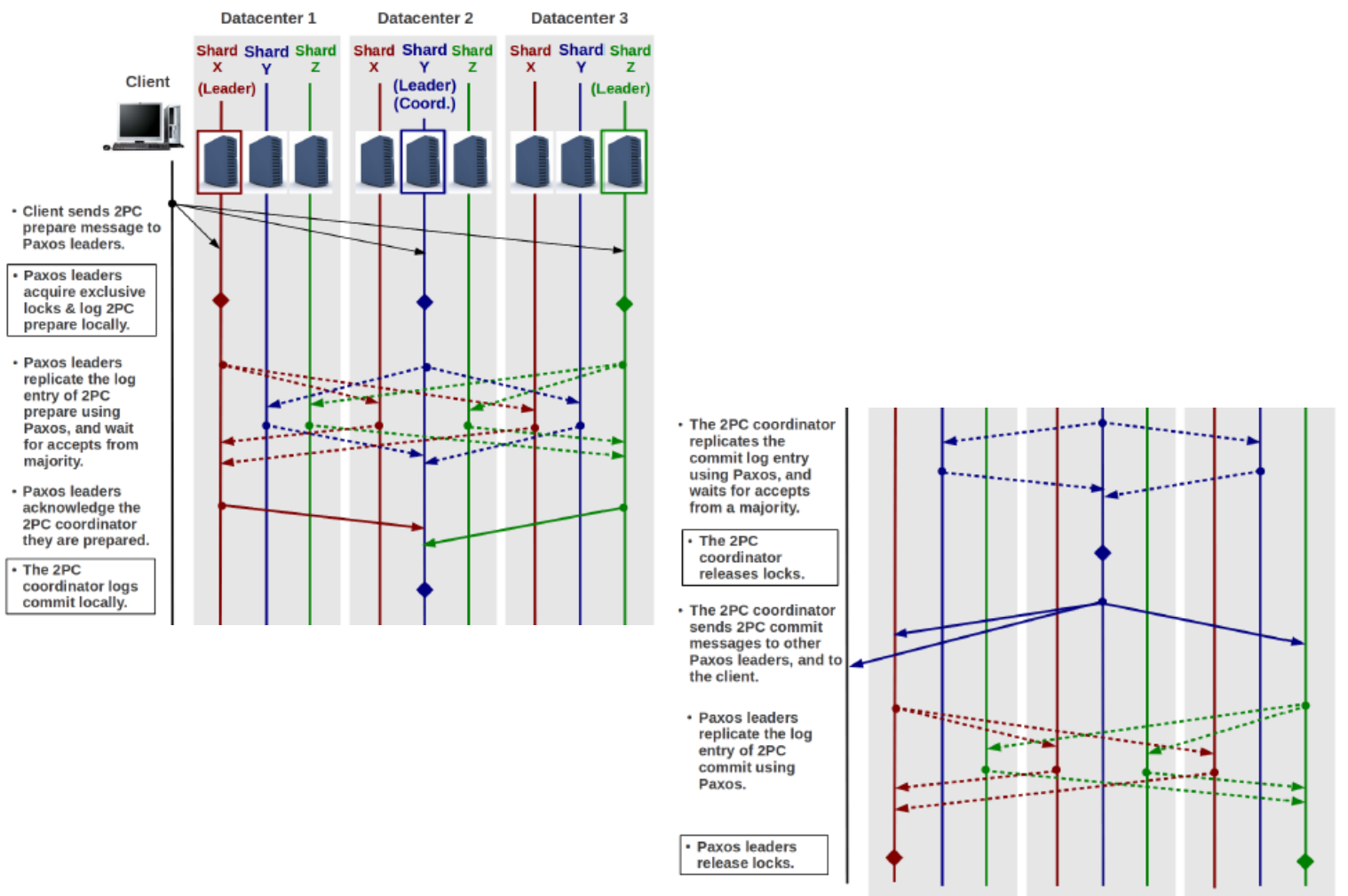


## Geo-Replikation für Cloud-Daten

- mehrere geographisch verteilte Data Center mit replizierter Datenhaltung (meist 3-5 Replikate pro Objekt)
  - schnelle Zugriffe auf räumlich nahe Replikate
  - Gewährleistung einer sehr hohen Verfügbarkeit gegenüber Ausfällen innerhalb Data Center sowie Katastrophen
- Beispielansätze: Cassandra, Yahoo Pnuts, Google MegaStore / Spanner
- meist synchrone Replikation („sehr sicher“) für Änderungen, um Datenverlust zu vermeiden
  - Google nutzt dazu Paxos-Verfahren zur konsistenten Aktualisierung der Replikate, eingebettet in 2PC
  - Beispiel für Client-initiiertes 2PC für Änderung von dreifach replizierten Objekten X,Y,Z (dunkel: Paxos Leader pro Replikatgruppe)



Quelle: D. Agrawal et al.:  
Managing Geo-replicated Data in  
Multi-Datacenters. LNCS 7813, 2013



## Replikation in Parallelen DBS

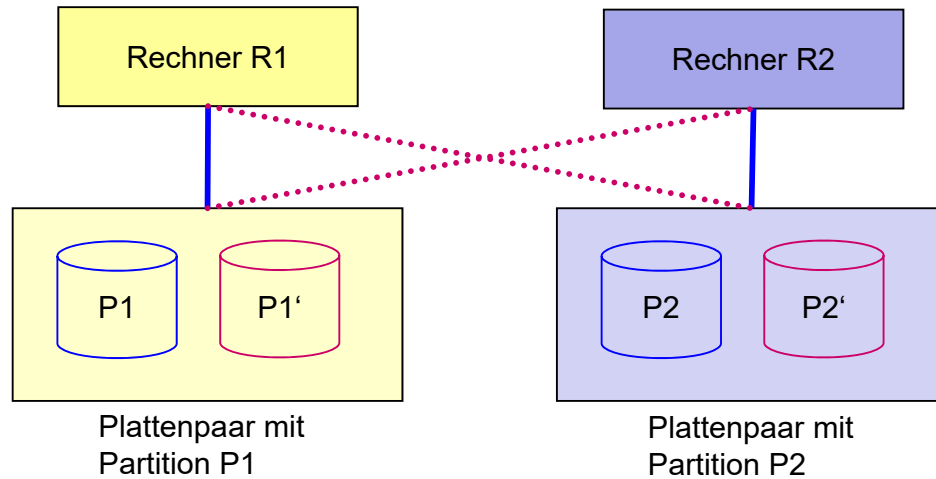
- replizierte Zuordnung von Fragmenten im Rahmen der Datenallokation
- Ziele einer Replikation
  - Fehlertoleranz gegenüber Externspeicherfehlern
  - Fehlertoleranz gegenüber Rechnerausfällen
  - günstige Lastbalancierung im Normalbetrieb und Fehlerfall
- Lokalität und Knotenautonomie weniger/nicht relevant
- i.a. synchrone (parallele) Aktualisierung der Replikate
- drei Replikationsansätze mit doppelter Speicherung der Daten
  - Spiegelplatten
  - Verstreute Replikation (NCR Teradata)
  - Verkettete Replikation (Gamma-Prototyp)





# Spiegelplatten

- weitverbreitet zur Maskierung von Plattenfehlern
- verbesserte Lese-Performanz
- Shared Nothing: Replikation auf einen Knoten beschränkt

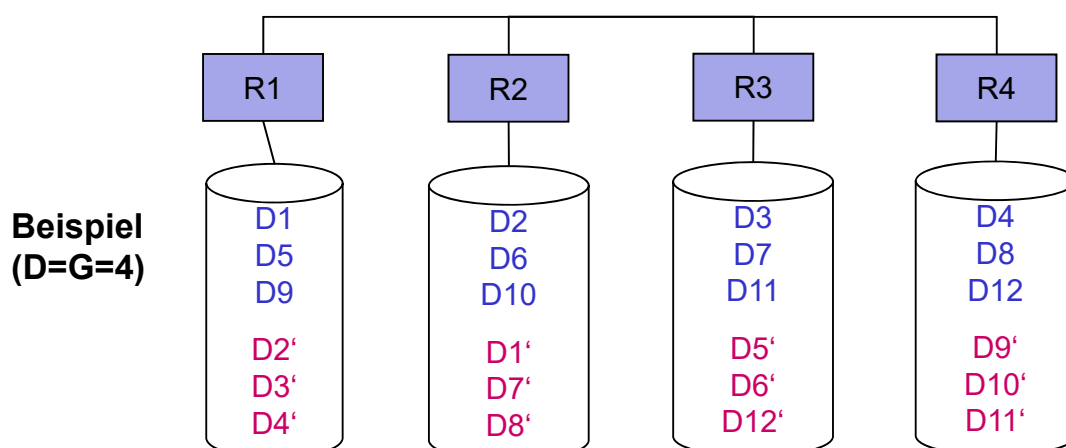


- Rechnerausfall erfordert Übernahme der kompletten Partition durch zweiten Rechner  
=> sehr ungünstige Lastbalancierung im Fehlerfall



## Verstreute Replikation (Interleaved Declustering)

- Ziel: bessere Lastbalancierung im Fehlerfall
- Datenknoten einer Relation werden in Gruppen von je G Rechnern unterteilt (node groups)
  - Replikate zu Daten eines Rechners werden gleichmäßig unter den G-1 anderen Rechnern der Gruppe verteilt
  - nach Crash erhöht sich Last jedes überlebenden Rechners der Gruppe gleichmäßig um Faktor  $G/G-1$



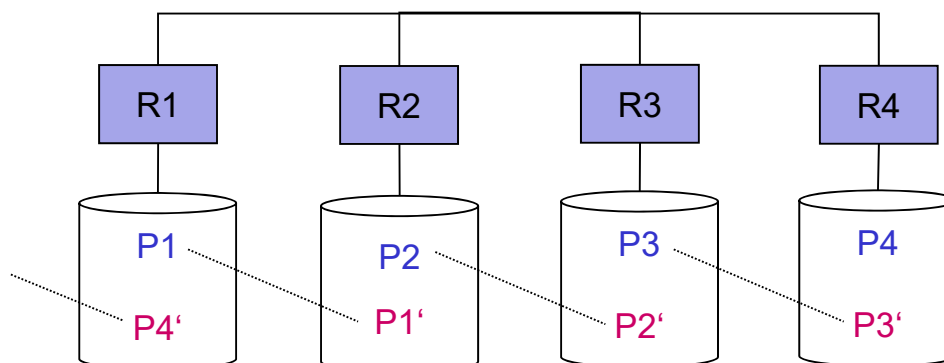
## Verstreute Replikation (2)

- Ausfall mehrerer Rechner beeinträchtigt Datenverfügbarkeit nicht, sofern verschiedene Gruppen betroffen sind
- Wahl von  $G$  erlaubt Kompromiß zwischen Verfügbarkeit und Lastbalancierung
  - Extremfälle:  $G=D$  (= Verteilgrad der Relation) und  $G=2$  (Spiegelplatten)
- Nutzung der Replikate zur Lastbalancierung im Normalbetrieb aufwendig



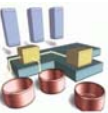
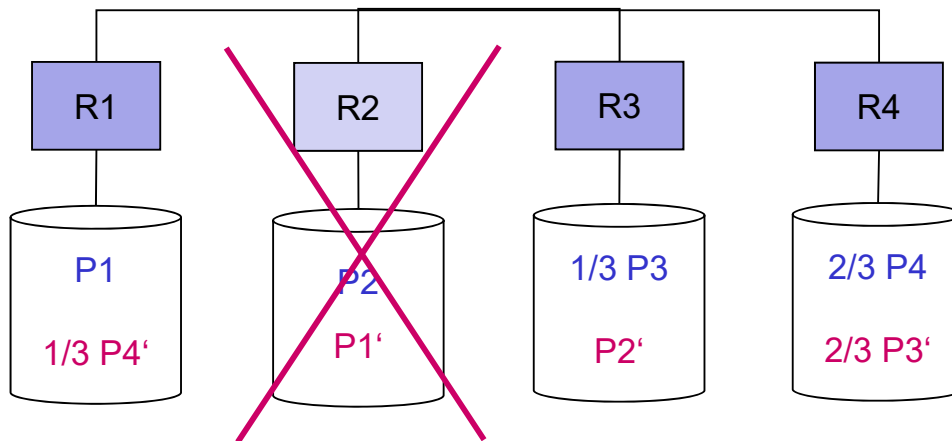
## Verkettete Replikation (Chained Declustering)

- Ziele: hohe Verfügbarkeit von Spiegelplatten + günstige Lastbalancierung (im Fehlerfall) der verstreuten Replikation
- Kopie der Partition von Rechner  $R_i$  wird vollständig am "nächsten" Rechner  $R_{(i \text{ MOD } G) + 1}$  der Gruppe gehalten
- selbst Mehrfachausfälle in einer Gruppe erhalten die Verfügbarkeit aller Daten, so lange nicht zwei benachbarte Rechner ausfallen



## Verkettete Replikation (2)

- Knotenausfall: Zugriffe auf betroffener Partition sind vollständig vom "nächsten" Rechner der Gruppe auszuführen
- Zugriffe auf den anderen Partitionen sind unter den beiden Kopien umzuverteilen, so daß Lastbalancierung erreicht wird
- keine Umverteilung von Daten, sondern nur Anpassung der Verteilungsinformationen



## Zusammenfassung

- Zielkonflikte: Leistung vs. Verfügbarkeit für Leser vs. Schreiber / Konsistenzanforderungen
- Synchronisationsansätze: ROWA, Voting, Primary Copy
- Probleme bei 1-Kopien-Serialisierbarkeit
  - synchrone Aktualisierung ist aufwändig (Commit-Protokoll)
  - geringe Skalierbarkeit
- kommerzielle DBS unterstützen unterschiedl. Replikationsmodelle
  - Publish/Subscribe-Ansätze, Primärkopien vs. „Update Anywhere“
  - synchrone oder verzögerte Übertragung von Änderungen
  - einfache Kopien und abgeleitete (transformierte) Replikate
  - parallele Änderungen mit Nutzerkontrolle (Merge-Replikation)
  - Parallele DBS: Duplizierung mit synchroner (paralleler) Aktualisierung
- Cloud Data: Geo-Replikation über mehrere Data Center
  - „billige“ Rechner mit hoher Ausfallwahrscheinlichkeit erfordern sichere Protokolle gegen Datenverlust, z.B. mit 2PC und Paxos

